Distributed Systems Project
Manal El Karchouni - Carmen Diez - Yong Jin Cheng - Alejandro Bravo de la Serna

UNIVERSITY OF HELSINKI

Distributed Analysis of a Dataset

[Github repository](#) - [Video demonstration](#)

Table of contents

I. The project's goal and core functionality.

Distributed computing is a model where the components of a system are spread out across multiple computers, but they run as one system. This approach allows to acquire and analyze information from Big Data. Different aspects of the distributed computing paradigm resolve different types of challenges involved in Analytics of Big Data.

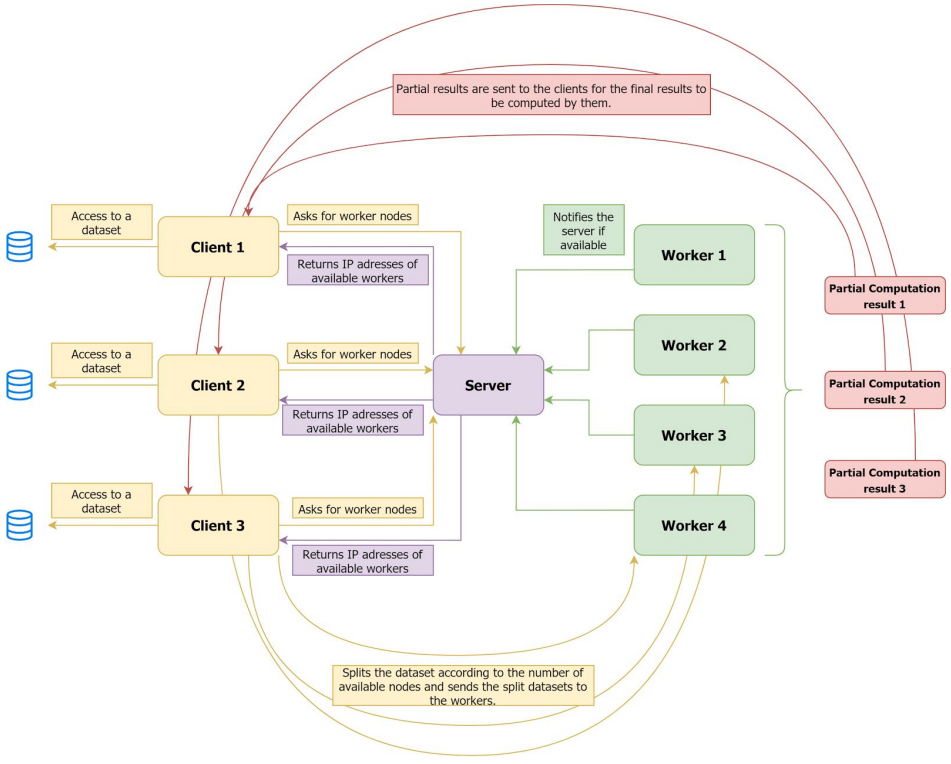
Our distributed system serves as a tool for distributed analysis of specific datasets. Simply stated, it consists of distributed autonomous nodes that communicate only over a network and it is a method of running programs achieving higher performance in both scientific computing and more "general-purpose" applications.

The main reason behind the concept of distributed analysis is the fact that heavy datasets, in terms of volume, velocity or veracity, require efficient and well-performing computing systems. Hence, distributing the analysis over different nodes is highly useful. There are many reasons to adopt this approach, such as increasing the performance, optimizing the latency, the availability of computers to connect, fault tolerance and sharing of resources, etc. By connecting several machines together, more computation power, memory, and I/O bandwidth can be accessed.

In our case, we have implemented a fault tolerant, synchronized system that is able to divide a dataset between several machines to split the work and ease the analysis. Right now the system supports the calculation of the average of a list of numbers, but other features could be easily included, as the distributed system basis is the most important part.

II. The design principles (architecture, process, communication) techniques.

Here is a simplified diagram of our architecture:



As presented in the diagram, our application is divided in three principal actuators: *client*, *server* and *worker*.

1. Client

The client first accesses the dataset he wants to do a specific computation operation on. He then initializes a connection with the server and asks him for a number of working nodes.
After receiving the IP addresses and the port numbers of the available working nodes from the server, the client splits the dataset accordingly and sends the subsets of the dataset to the available nodes. If a working node crashes in the process, the client is able to recover the data he sent to it and then sends it to another node.

2. Server

The server has the main role of managing a queue of the working nodes (and the not available ones) and establishing a link between the client(s) and the worker(s). He's in charge of :

- Sending N working nodes to the client where N is the number of working nodes that the client asked for or less (if not that many are available).
- Sending more available workers to the client if one of them crashes in the process or if there weren't enough available workers at the beginning.
- Identifying the nodes that are no longer working and those that are not available.

3. Worker

Each worker node is in charge of receiving data subsets from the client and processing them. The workers keep in contact with the server to inform him about their state, sending different messages depending on it:

- 'Yes': when they are available and ready to receive and process data. Workers send it at the beginning and also when finishing some computation. When the server receives this message it adds this node to the list of available ones.
- 'No': when they are busy, in this case, when they receive some client's data. When the server receives this it adds the node the not ready workers list.
- 'PING': that is sent from time to time to the server. This way the server knows if a node is still alive.

The workers are able to process a partial result for the final computation. When they are done processing the data they send it back to the client so that it computes it as it wants.

III. The key enablers and the lessons learned during the development of the project.

1. Lessons learnt

The main lesson learned during the development process of this project is **building a simple distributed system running on at least three nodes**.
First, the focus was on establishing an architecture that not only will allow us to develop a first version of our system (one client and one worker), but also to scale it in order to **support an increased number of nodes and of clients**.
Another concept that we learned from our project is ensuring a **synchronized communication between the three main components of our architecture** (the client, the server and the worker) and **managing a queue of working nodes** according to their availability.
We were also able to implement a system that is capable of **recovering when a node goes down**. This has to be done to ensure that the distributed system is still operational when one of the nodes crashes and that it does not break the whole process.

2. Technical challenges

The project has also been a source of challenges. Technical bugs kept showing up and we had a hard time identifying their source. For example, we had problems with *pickle* and *json* library. For some reason, when sending a pickle/json data, sometimes it got mixed in two. Finding these issues was challenging because when we debugged we assumed that library calls were operating correctly. To fix it, we parsed the json data and divided it into its corresponding individual jsons. This has shown us that unexpected challenges can always appear. And, in general, that many problems come attached to synchronization and working for the same process in different computers.
Another challenge was making decisions regarding some important metrics of our system. For example : the time it takes for a client to recover from a faulty node. Deciding on the value of this time was actually a difficult decision, as having a greater time would slow the process a lot, but with a lower time we would need a lot of interaction with the server to not declare nodes dead when they are not.

IV. How do we show that our system can scale to support the increased number of nodes?

Our system currently supports running multiple client and worker nodes, the client(s) can split up the dataset based on the number of worker node(s) requested and are currently available then send them to the worker node(s). Hence, if our system has more clients we can easily accommodate it by adding more workers to the network, speeding up the computation.

Furthermore, even though our system only supports running one server that is responsible for coordinating between clients and workers, we believe that by adding more servers to the network we could also greatly improve the performance of our system (in case one of the server fails or to be able to attend more clients).

V. How do we quantify the performance of the system and what did we do (can do) to improve the performance of the system ?

We have chosen to quantify the performance of our system by measuring the average task completion time of our system. We defined our task completion time as t**he time taken for our client to successfully send the data to the worker node(s) and receive the result**. We had measured the effects of increase in the number of worker nodes for a set of data of the same size and the increase in the size of the data processed. Our testing dataset consists of a list of size $10^{(1 \text{ to } 7)}$ integers, our worker node(s) would compute the mean of the list sent to them and return it back to the client. Below is the result of the average task completion time differed by number of workers and size of the dataset.

Array size | Number of workers	1	2	3	4	5	6	7	8	9	10
[10]*10	0.004987	0.001995	0.002987	0.003739	0.003837	0.003963	0.004769	0.004837	0.004957	0.005031
[10]*100	0.09274	0.053583	0.029681	0.024785	0.032943	0.029646	0.025783	0.024951	0.022666	0.022708
[10]*1000	0.012969	0.007697	0.004851	0.006755	0.004839	0.005832	0.005984	0.00773	0.006746	0.025728
[10]*10000	0.094766	0.038913	0.038898	0.026945	0.032909	0.027946	0.026795	0.024687	0.022896	0.02167
[10]*100000	0.726036	0.443837	0.32201	0.284236	0.2351	0.228159	0.201314	0.192357	0.182278	0.171537
[10]*1000000	7.125965	4.368322	3.239107	2.649916	2.364475	2.147294	2.016498	1.875834	1.762021	1.682346
[10]*10000000	74.94675	40.42193	28.54845	23.05239	20.31171	19.24255	18.55503	18.20133	17.539	16.82786

We can see that there is little to no effect in the average completion time for an array of size less than 10000. However from an array of size 100000 onwards, you can see a significant improvement in the performance evidenced by the decreasing task completion time.

We might still be able to further improve our result by including more servers to delegate the workloads, since there might be too many requests for just one server.

VI. What functionalities does the system provide?

1. Node discovery

Our system supports node discovery functionality. The server initiates a worker socket, which is in charge of listening to the workers. The worker node accepts the connection to the server and sends the “Yes” message we explained before to it, indicating that it is now available and ready to process. When receiving this message, the server moves this node to the list of the available ones and this is how a node is discovered.

2. Fault tolerance

A key functionality of our system is fault tolerance. This means that our system is able to recover from partial failures without affecting overall performance. In our case, a partial failure would be one of the nodes going down after it's been made in contact with the client and during the process of the data.

First, as mentioned, each node periodically sends a "Ping" message to the server to let him know that it's still working. If it's been inactive for too long, the server declares it as "dead". We decided that a node is considered dead if there is no interaction with the server in more than 10 seconds.

When that happens, the server notifies the client of this inconvenience and sends him another working node, if available. The client then is able to recover the data subset that he sent to the dead node, in order to send it to the new one.

3. Synchronization

The actuators act synchronized between them as they wait for each other's information, especially for the client and the others in the network. First, the client would contact the server asking for worker nodes, and it must wait until it receives some worker nodes from the server to begin sending the data. Moreover, as the server might not send all the requested workers at once, the client must keep in contact with the server in the cases that it sends more nodes or notifies the client of the dead workers. Also, after sending the data to the worker, the client must wait for the workers' answers to finish before it can leave. Also, there is a short time required to declare a worker as 'dead', so if one of the nodes fails, the client must wait for at least 10 seconds before getting notified of the particular dead worker from the server and resending the data to another available worker again.